Contemporary Engineering Sciences, Vol. 10, 2017, no. 15, 739 - 750
HIKARI Ltd, www.m-hikari.com
https://doi.org/10.12988/ces.2017.7876

A Data Structure to Represent Data Sets with More Than One Order Relation like Polygons

Edgardo Samuel Barraza Verdesoto

Corporación Unificada Nacional (CUN), Colombia

Edwin Rivas Trujillo

Universidad Distrital Francisco José de Caldas, Bogotá, Colombia

Duván Cardona Sánchez

Pontificia Universidad Javeriana, Colombia

Abstract

This paper introduces a new data structure useful to represent objects or data sets with several order relations between their elements. A specific case is the polygons that will be analyzed in this paper. Polygons are objects often used in Computer Graphics and their operations have a high computational cost; this data structure was implemented in Java language with a performance *O(n*log(n))* corresponding to building and *O(log(n))* to searching. The complexity analysis for *insertion* and *searching* routines are explained.

Keywords: Data Structure, Order Relation, Polygon, Complexity Time

1 Introduction

There are data sets with elements that can establish several order relations between them. Some of these objects are the intervals. This paper introduces a new data structure that models this kind of sets from the point of view of their order relations by reducing the computational cost in their basic operations.

The paper will be organized as follows: first, the motivations and the context where the main ideas arise from will be described. Second, some definitions will be provided and some data structures comparable with the proposal will be described. Third, the data structure and their implementation will be specified. Finally, a complexity analysis will be done and the conclusions will be drawn.

2 Motivation

This proposal arises from a project of image processing where it was required to continuously determine if a point was inside a polygon or not. The most popular algorithm for this proposal is based on the Jordan curve theorem through a complexity time *O(n)* [3].

Currently, the polygons are represented computationally by a point succession saved in a vector where each element is the next vertex. Any search would always require a complexity O*(n)* because this data structure has no order criterion.

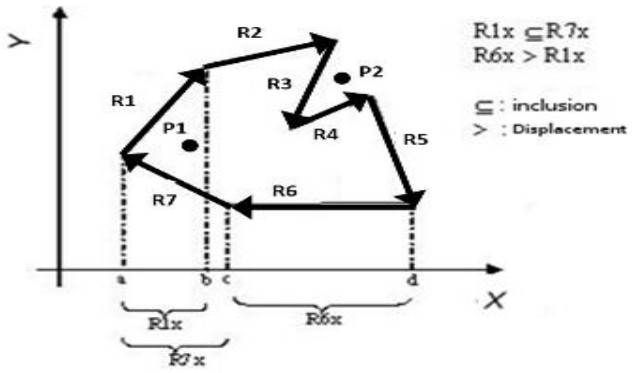

Fig. 1. Projection of a polygon on the X-axis.

The polygons can be decomposed into a couple of set of intervals which are the projection on each coordinate axis. Fig. 1 shows a polygon projected on the x-axis making intervals, where both ***R1x*** and ***R7x*** are displaced both with regard to ***R6x*** and ***R1x*** is included in ***R7x***. Now, if a set of data is ordered, it is easy to find a key on the inside, but ordering is a preprocessing. *Space* partitioning is a method used in Computer Graphics for preprocessing the Euclidean space by dividing it in small pieces like rectangles stored in a searching tree. This technique has played an key role to improve the performance in operations like ray tracing, and it have generated several proposals of data structures very powerful [4].

The next idea was to build a data structure with an insertion procedure like the *insertion sorting algorithm*. Implementing a structure sorted under two orders is a complex task because it is necessary to build and to dynamically link structures and no nodes as commonly is accustomed. These tasks will be explained further on. The great advantage of this strategy is to avoid the preprocessing and to extend the

abstraction of a polygon. The complexity time to build these sort of data structures is *O(n*log(n))*, and *O(log(n))* for searching.

3 Definitions

A. Order relation

If the elements of a set could be compared according to common properties that identify their magnitude between them, it is said that the set is ordered, and the specification on how to compare among themselves is named *Order Relation* [5]. Order relations could be classified into *Partial, Total,* and *Quasi-Order*. The first two meet the properties *reflexive, antisymmetric,* and *transitive*; the difference is that the second satisfies $x\mathbf{R}y$ or $y\mathbf{R}x$. The last are attributable to the *irreflexive and transitive* properties [5].

B. Double order relation

When comparing two elements of a data set and it is possible to establish two different kinds of order relations, it is said that the data set has a Double Order Relation. For example, the intervals are attributable to two order relations: *Partial order* (Inclusion) and *Quasi-order* (Displacement).

4 Double Order Structure, Description and Composition

A data structure is a way to organize the information in a computer to be used efficiently. They can be used to arrange elements under an order criterion, but it must be only one because the conventional data structures are not able to classify elements with intrinsic several orders like the intervals. Segment trees and interval trees are tries to represent intervals through data structures, but they don't respect their implicit order relations [7].

Before describing how a data structure with two orders would be, a sort of list called *skiplist* will be explored; it has some characteristics advisable in a structure ordered under two different criteria.

A. Linked List with Skips (Skiplist)

It is a variant of the linked list composed of several lists organized in layers which have forward skips defined randomly. Each subsequent layer points elements from the last layer with a probability *p*. The last (or first) layer points all the elements. A search begins in the highest (or lowest) level passing through the next levels until it finds the key. If the list has a moderated number of levels, the search could be done in $O(Log(n))$ [8]. Fig. Nº 2 shows a *Skiplist* in which novel items are added.

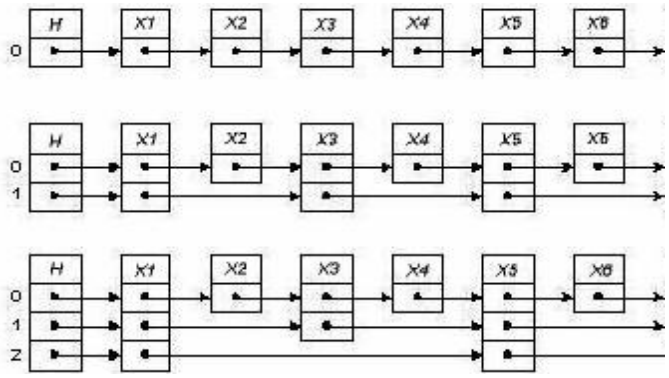

Fig. 2. A *skiplist* with three levels; each node has a pointer towards the next node in accordance with the skip determined in the level.

B. Double Order Structure (DOS)

It is a structure that organizes the elements contained in it under two different orders. Fig. 3 shows two of these structures corresponding to the intervals projected on the coordinated axes by the polygon in Fig. 1. The horizontal nodes are ordered by *Displacement* and the vertical ones by *Inclusion*. Like the *Skiplist*, these structures grow in levels but by nodes which can contain zero or multiple levels. The top level is called the *root-level*.

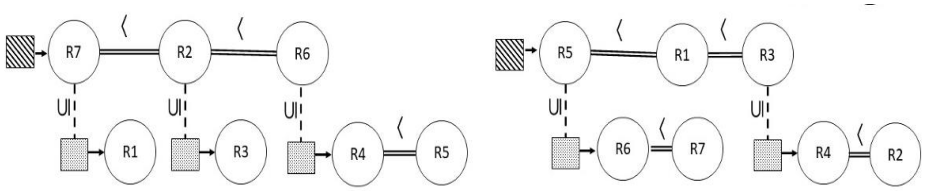

Fig. 3. Two Double Order Structures representing the intervals projected by the polygon in Fig 1 for each coordinated axis: X (left image) and Y (right image). Both have two views, horizontally with Quasi-order and vertically with Partial Order.

5 Implementation

A *Double Order Structure* (**DOS**) is composed of three elements: a *flag*, a *Node-Array* and a *DOSPointer-Array* (see Fig. 4). The first element detemines the kind of order relation in the level, the second contains a list of keys ordered under the relation specified by *flag*, and the third is a pointer towards an array of DOS's.

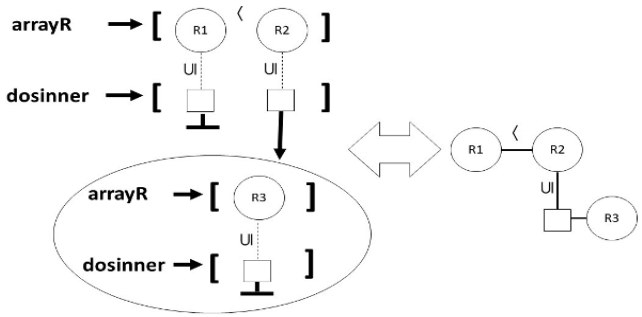

Fig. 4. Anatomy of a *Double Order Structure*.

The ADT in Fig. 5 shows the *flag* modelled by *boolOrder*, the *Node-Array* by the *arrayR* array, and the *Pointer-Array* by the *dosinner* array. The implementation introduces the variable *AB* which is the scope of the keys in *arrayR* and is interpreted as the level scope. Each *dosinner[I]* represents a *set of levels* under *Node-Array[I]* with an order criterion opposite to its instance. Note that each *dosinner[I]* is a linked list of structures ordered by inclusion. Other variables that appear were used for the specific purposes.

```
public class DoubleOrderStructure
{
   private double[] AB;

   private GenArrayList<Integer> arrayRId;
   private GenArrayList<Line2D> arrayR;
   private GenArrayList<DoubleOrderStructure> dosInner;

   private final boolean boolOrder;
   private final int intCoordinate;

   public DoubleOrderStructure(int C, int ID){}
   public DoubleOrderStructure(boolean boolO, int C, int ID){}
   public int addNewLine(Line2D lineNodeData, int intId){}
   public GenArrayList<Line2D> srchLinesNearestPoint(int[] intP){}
   public GenArrayList<Line2D> srchLinesAbovePoint(int[] intP){}

   //Binary Search of a line in a set ordered by displacement: O(log(n))
   private int[] binarySrchLFO(Line2D lineData){}

   //Binary Search of a line in a set ordered by inclusion: O(log(n))
   private int[] binarySrchLSO(Line2D lineData){}

}
```

Fig. 5. Abstract Data Type (ADT) of the Double Order Structure implementation.

The DOS implementation included *insertion* and *searching* operations, except for *remove* because it was not necessary for the application; however, it can be demostrated that this operation could be implemented with good performance. The core of the implementation was the *search*, for which a couple of variations of the *binary search* were built adequate for intervals.

A. Adding a New Interval

Suppose there is a DOS where every *arrayR* are under *Displacement* (horizontal order), and all nodes of *dosinner* are under *Inclusion* (vertical order). Each time one key is inserted, the method *addNewLine* is invoked; that, in turn, calls *binarySrchLFO* which is the implemetation of *binary search* algorithm for intervals ordered under *Displacement*. The search returns a pair **<code-op, index>** determining one of the following possibilities:

- *<0-insertion, index>*: No interval includes the interval processed.
- *<1-included, index>*: One interval includes the interval processed.
- *<2-inclusion, index>*: There is one or more intervals included by the interval processed.

<0-insertion, index>. It occurs when the index indicates where the new element should be inserted. Fig. 6 (steps 4 and 5) shows the insertion process of a new key into the DOS-Y structure where the index results are equal to -1, then R5 will be inserted as a new header of the top level.

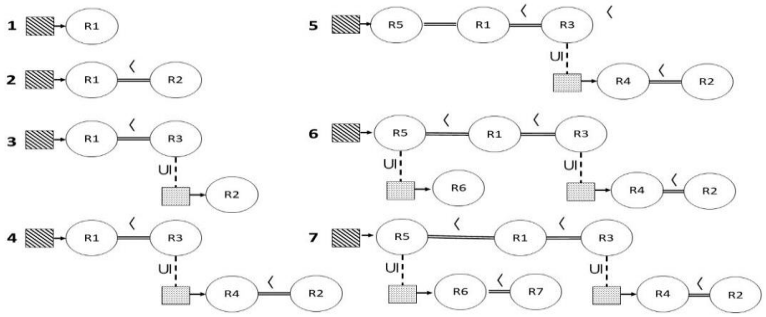

Fig. 6. Inserting keys into the DOS-Y structure. This operation is based on the polygon in Fig. 1.

<1-included, index>. Supposing this possibility happens, a key in the current level will include the interval to insert. Let *I* be the index resulting, then the key located in *arrayR[I]* includes the new interval, and *dossinner[I]* would be a pointer to a structure with levels ordered by *Inclusion*. In this state, the new element is moved to *dosinner[I]* and the routine *binarySrchLSO*, associated with this structure, is executed. *binarySrchLSO* is the implementation of *binary search* algorithm for intervals ordered under *Inclusion*.

binarySrchLSO routine determines the level where the new interval should be inserted. It is important to mention that *dosinner[I]* is ordered from the greatest to lowest domain, thus, *dosinner[I][K]* includes all the keys belonging to levels greater than *K*. If there is any level that includes the item, the result will be an index *J* which corresponds to the top level that includes that item, then this will be inserted into the structure *dosinner[I][J]*. The next step consists in moving the new element

to the structure pointed by *dosinner[I][J]* with another called to *addNewLine* in this context (see Fig. 6, steps 3 and 4).

If the result of *binarySrchLSO* is an index greater than *dosinner[I]* size, then a new level will be created at its tail by calling *addNewLine* in this context. *binarySrchLSO* checks two special cases before carrying out its process. The first verifies if the scope of the head (*AB* variable) is included in the new element; then a new level is created at the top of *dosinner[I]*, that is, it will be the new head. The second case verifies if the new element is *displaced* with regard to the scope of the head (*AB* variable); then the new element is inserted in the head (*dosinner[I][0]*) (see Fig. 6, steps 5 and 6).

Note. If any structure pointed by a *dosinner[I][J]* runs an *addNewLine* operation, this will rebuild the keys under *Displacement* (see Fig. 6, steps 3 and 4).

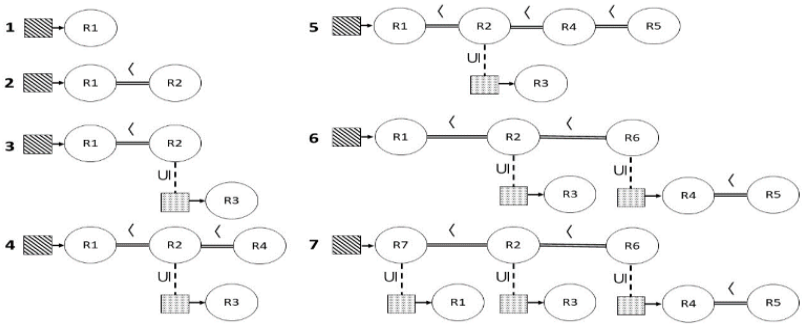

Fig. 7. Inserting keys into the DOS-X structure. This operation is based on the polygon in Fig. 1.

<2-inclusion, index>. This special case occurs when the new key includes one or more intervals of the current level. The operation requires two steps. The first is to replace all intervals included by the new key and then move them to a new level created under the new key. Fig. 7 (steps 5 and 6) shows an example where *R6* interval is added into the *root* structure but the *R4* and *R5* intervals are included inside it.

B. Verifying if a point is inside a polygon (ray tracing method).

The *ray casting algorithm* [1] states that if a line traced from a point intersects an even number of sides of a polygon, then the point is outside it. The implementation was performed following these concepts, but it was not necessary to make the intersections with all sides of the polygon, not even in the worst case.

The routine *searchLinesAbovePoint* finds all the intervals intersected by an "imaginary" vertical line (using DOS-X structure) or a horizontal line (using DOS-Y structure). *searchLinesAbovePoint* results in an index *I* on the root level as a

starting point to search nodes forwards and backwards, in *arrayR*, that include the point. For example in Fig. 1, if the point is in middle of **b** and **c**, the result will be 0 which is the index of **R7** in *arrayR*; then an *expansion* operation that includes **R2** will be executed. For each key found, their levels will be explored for searching new intervals which could include the point.

During the process, the intervals above the point (or to the right, with DOS-Y) are counted, according to the ray casting algorithm [1]. It is known that this method has several problems with the vertex, but this implementation overcomes them.

C. Verifying if a point is inside a polygon (novelty method)

This method requires taking into account the sense used to create the polygon. Fig. 1 shows that the polygon is created clockwise. According to this, whatever point located on the right of any side will be inside the polygon. Thus, it is enough to find the nearest line to the point and verify if the point is located on the right side or on the left side.

The routine *searchNearestPoint* was built to find the two nearest lines to the point: below and above (left and right hand using DOS-Y), but it would be enough with one of them. The program looks for keys using *binary searching* like *searchLinesAbovePoint*. The expansion process is also executed but the process is less intensive because *searchNearestPoint* only calculates the distance from the point to the line, discarding or replacing the current.

6 Complexity Analysis

A temporal analysis on the operations described before was done. These operations are: *addNewline*, *searchAboveLinesPoint*, and *searcNearestLinePoint*. The former is, technically, an insertion operation and the two latter are used to look for keys.

A. Insertion process (addNewline)

Fig. 7 shows step by step how the structure DOS-X is built. Initially, it is empty, then R1 is inserted in the root and the subsequent insertions looks for keys included in the new element or vice versa; this process is executed by binary searching. If the novel item contains some intervals of the structure, an *expansion* process is executed (see last section) to determine what additional adjacent intervals are included in this item. The worst case can occur only one time: when all the intervals are ordered by *displacement* and the new key contains all of them, usually in convex polygons.
More formally, it will imply that each structure by level is under *Displacement order* and the levels under *Inclusion*, there are k keys (lines) in each level and b levels for each key with a total number of keys $N= k*b*k*b...$; then a new key would be inserted by searching the correct level.

The procedure for knowing if a *key* could be inserted in some intervals over any level is executed in *O(log(k))*. If any interval includes the new key, then the control should be moved to the structure which is attached to this node; the node can have several levels by *inclusion order*. The process to locate the level where the key could be inserted is done in *O(log(b))* time. This process continues over other structures selected until the key is inserted. Supposing that the process is homogenous, the time will be:

$$O(log(k))+O(log(b))+O(log(k))+O(log(b))....< O(log(N))$$

The "**less than"** inequality is because this procedure does not examine all levels. If the number of keys to insert is N and the procedure, in general, is as described, the complexity time will be **less than** *O(N*log(N))*.

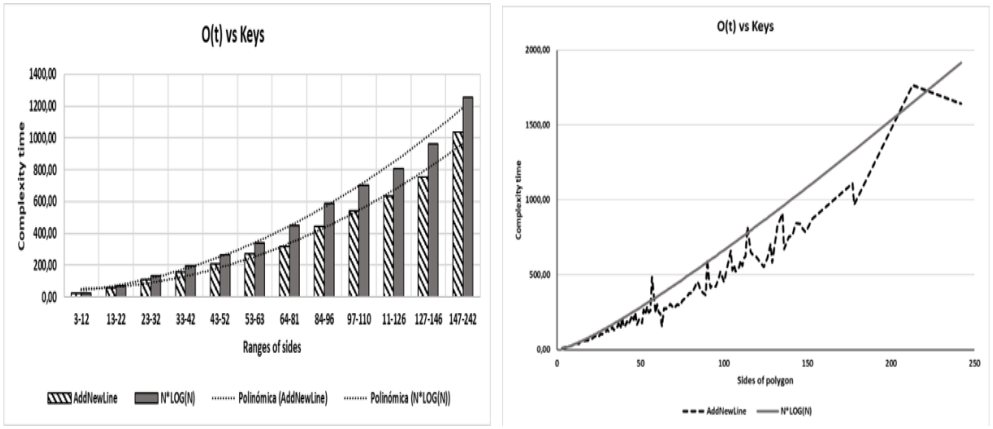

Fig. 8. Complexity time comparison between the insertion procedure *addNewLine*, applied over several polygons, and *O(n*log(n))* time.

Fig. 8 shows the complexity time of the insertion procedure calculated for 684 polygons. The left graph is presented with the samples grouped by their sides. The *O(n*log(n))* time was chosen as the ceiling for this algorithm because it is the common limit in preprocessing polygons [2][3][6]. It is important to explain that, although the curve is soft, it is not true completely because the algorithm in the expansion procedure can have an *O(k)* or *O(1)* time. The curve without ranges can be observed in the right graph of the Fig. 8, that shows the peaks created by this behavior.

To summarize, the structure has two important characteristics: an *ordered insertion* very ideal for searching before insertion, and a *good distribution of the keys* over all the structure guaranteeing a superior performance in the expansion procedure.

B. Searching process (searchLineshAbovePoint and searchLinesNearestPoint)

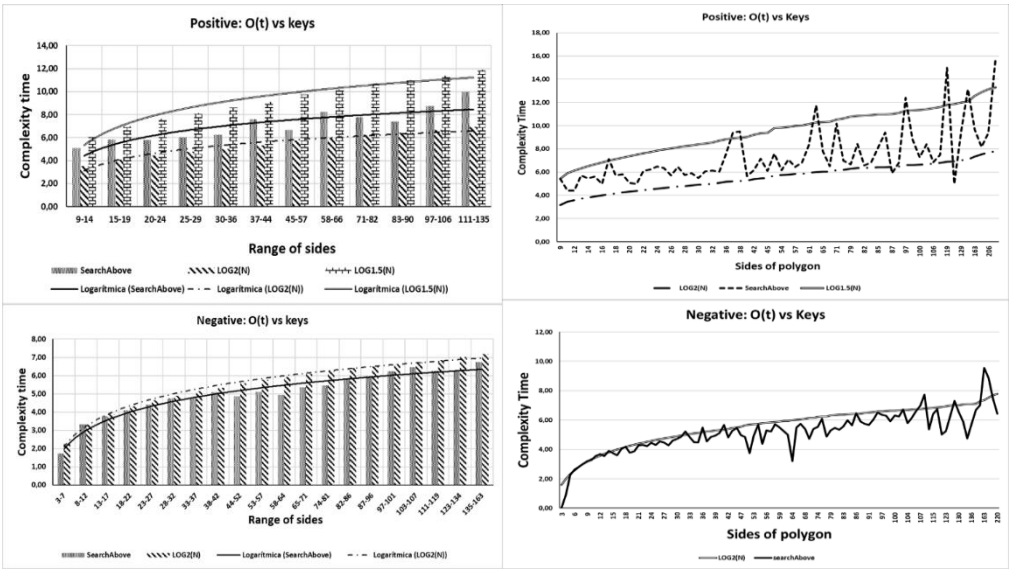

Fig. 9. Complexity time comparison between the searching procedure *searchLineshAbovePoint*, applied over several polygons, and *O(log(n))* time.

There are algorithms which are used commonly to search points in polygons that require preprocessing. Those usually take *O(n*log(n))* time for preprocessing and *O(log(n))* for searching [2][3][6]. The methods proposed here do not need preprocessing because the intervals that compose the polygon are sorted immediately are inserted. As *searchLineshAbovePoint* and *searchLinesNearestPoint* routines are carried out, they function like the common *binary search* but applied to *intervals*, the process has the following complexity:

$$O(log(k))+O(log(b))+O(log(k))+O(log(b))....< O(log(N))$$

Two classes of experiments were performed by sort of search; for positive and negative answers. The tests to verify *searchLineshAbovePoint* were done over 2,419 cases where a point was inside a polygon (P-case), while the negative case (N-case) took place over 369,077. Fig. 9 show the results, The P-case behavior is in middle of two logarithmic curves: Log_2 and $Log_{1.5}$. The N-case is under Log_2. The left graphs of the Fig. 9 reveal the experimentation by grouping ranges of polygon sides and the mark is the average of this range. On the other hand, the right graphs of the Fig. 9 show this testing in terms of the polygon sides where it is possible to observe the peaks because the *expansion* behavior.

The experiments to verify *searchLineshNearestPoint* is like the one before. It was performed over 1,483 cases where a point was inside a polygon (P-case) and over 323,909 cases for N-case. It revealed that the method requires fewer steps in its performance for both cases. Fig. 10 shows the results grouped by range of sides (left graphs) and by sides number (right graphs).

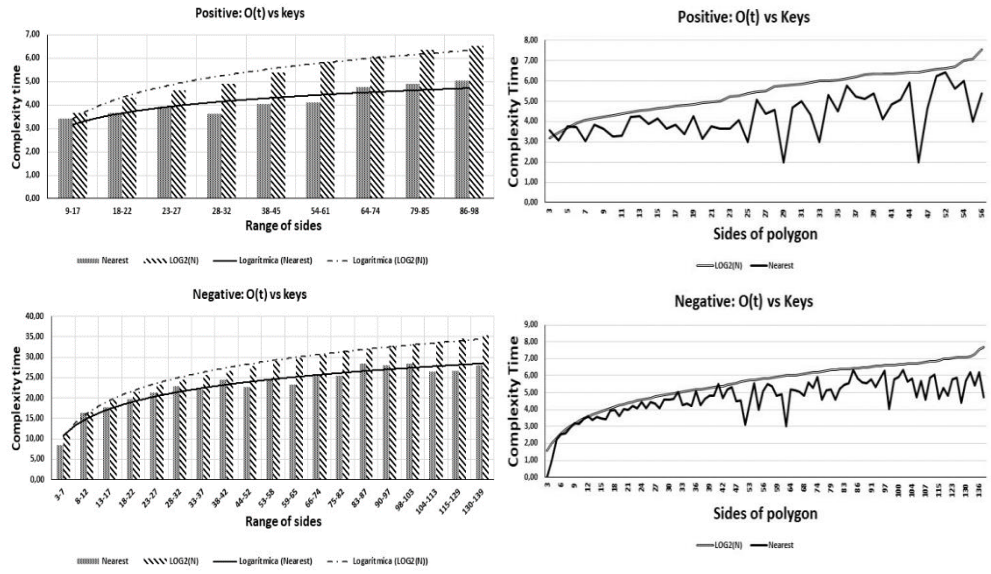

Fig. 10. Complexity time comparison between the searching procedure *searchLineshNearestPoint*, applied over several polygons, and *O(log(n))* time.

7 Conclusions

This paper explained a data structure which is a powerful tool to model the order relations embedded in several kinds of complex data. For example, polygons whose projection over the X or Y axes generate intervals that can be organized with two intrinsic order relations: *Inclusion* (*partial order*) and *Displacement* (*quasi-order*).

If the order properties of a data set are implemented in a computational model, their basic operations improve their efficiency. Proof of this are the *insertion* and *searching* algorithms, explained in the last sections, for which a structure was built (*Double Order Structure*) to store their elements under the two intrinsic order relations at the same time.

The *Double Order Structure* was used to store the sides of polygons projected over the axes keeping their order properties: *Inclusion* and *Displacement*. Although a polygon has two projections, only one of this is enough to know if a point is inside a polygon, as proved here. The data structure was implemented in Java and their complexity analysis is exposed in terms of size of the samples. Such proofs demonstrate superior performance with regard to other techniques, especially in searching.

Finally, the document introduces a new way to know if a point is inside a polygon which adapts perfectly to the order concept underlying in the structure and with superior performance.

References

[1] A.S. Glassner, *An Introduction to Ray Tracing*, Academic Press, 1989.

[2] D.G. Kirkpatrick, M.M. Klawe, R.E. Tarjan, Polygon triangulation in O(n log log n) time with simple data structures, *Discrete and Computational Geometry*, **7** (1992), 329–346. https://doi.org/10.1007/bf02187846

[3] E. Haines, Point in polygon strategies, Chapter in *Graphics Gems IV*, Academic Press, 1994, 24-46. https://doi.org/10.1016/b978-0-12-336156-1.50013-6

[4] G. Zachmann, E. Langetepe, Geometric Data Structures for Computer Graphics, *Proceedings of Eurographics 2002 Tutorials, The Eurographics Association*, (2002).

[5] K. Ross, C.H. Wright, *Discrete Mathematics*, Prentice Hall, 1990.

[6] R.E. Tarjan, Ch.J. Van Wyk., An O(n log log n)-time algorithm for triangulating a simple polygon, *SIAM Journal on Computing*, **17** (1988), 143–178. https://doi.org/10.1137/0217010

[7] T.H. Cormen, Ch.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 2009.

[8] W. Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees, *Communications of the ACM*, **33** (1990), 668-676. https://doi.org/10.1145/78973.78977

Received: August 18, 2017; Published: September 28, 2017